ESD-TR-75-81

FILE COPY

ESD ACCESSION LIST
XRRI Call No. 83206
Copy No. 1 of 2 cys.

RESEARCH IN PROGRAM OPTIMIZATION TECHNIQUES

President and Fellows of Harvard College
Cambridge, MA 02138

30 June 1975

Approved for public release;
distribution unlimited.

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
HANSCOM AIR FORCE BASE, MA 01731

ADA015041

LEGAL NOTICE

When U.S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

OTHER NOTICES

Do not return this copy. Retain or destroy.

This technical report has been reviewed and is approved for publication.

George E. Reynolds

GEORGE E. REYNOLDS
Techniques Engineering Division
Information Systems Technology
Applications Office

[illegible]

STANLEY P. DERESKA, Colonel, USAF
Chief, Techniques Engineering Division
Information Systems Technology
Applications Office

FOR THE COMMANDER

Frank J. Emma

FRANK J. EMMA, Colonel, USAF
Director, Information Systems
Technology Applications Office
Deputy for Command and Management Systems

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ESD-TR-75-81	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* RESEARCH IN PROGRAM OPTIMIZATION TECHNIQUES		5. TYPE OF REPORT & PERIOD COVERED 1 June 1974 - 31 May 1975
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Thomas E. Cheatham, Jr.		8. CONTRACT OR GRANT NUMBER(s) F19628-74-C-0208
9. PERFORMING ORGANIZATION NAME AND ADDRESS President and Fellows of Harvard College Cambridge, MA 02138		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy for Command and Management Systems Electronic Systems Division Hanscom Air Force Base, MA 01731		12. REPORT DATE 30 June 1975
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Structured Programming
Program Optimization
Implementation Languages
Register Assignment
Temporary Storage Minimization
Symbolic Evaluation
Program Metering
Common Expression Removal

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

In the context of the ECL programming system, general techniques for program optimization at high levels of language and special purpose techniques to enhance use of ECL for systems programming have been studied. The specific problems discussed are the efficient representation of knowledge about programs, the use of measurements to guide program improvement, compiler optimization under strict resource constraints and user control of machine level code optimizations such as register assignment.

I. Introduction

An objective review of program optimization technology has revealed a number of problems whose solutions are both essential (if new methods of organizing program development are to be successful) and relatively accessible today.

(a) Structured programming techniques emphasize modular program construction and well-defined layers of representation in order to minimize the cost of producing correct, reliable software. Often, however, the structure introduced for clarity inhibits direct translation of an abstract algorithm to an efficient machine language form. The trend toward structured programming increases the need for tools to merge isolated modules and to eliminate inefficient layers of structure in a cost effective way. If efficiency remains the price for clarity and maintainability, the new disciplines will not be fully accepted.

(b) Development of programs by stepwise refinement should be synonymous with program optimization. Program improvement aids should not be limited to the translation of source to machine code; they should be employed at each step from an abstract level of representation to a more concrete level.

(c) Optimizers are too often expensive, weakly governable engines whose workings are not easily grasped by users and whose effects are only indirectly observable through changes in the runtime behavior of compiled programs. While it should not be necessary to understand a tool in order to use it, the tool should nevertheless be precisely controllable and readily adaptable to accomplish special tasks. Its effects should be expressed in terms the programmer can appreciate, preferably a modified or augmented copy of his program. Where the user must take control of operator definitions and data representations at the hardware level, he should be able to do so without completely suspending optimization.

(d) It is now time for the results of research in the field of verification and automatic programming to begin affecting the design of practical tools. Current optimizers make too little use of knowledge deducible from programs or from measurements of their behavior, and are inflexible about accepting help in the form of assertions about the programs or their data. Too often the lack of a complete or efficient algorithm for a particular problem has inhibited the use of sound heuristics in critical regions.

Our approach to the preceding problems has been two-fold. In the context of the ECL programming system, we have examined (1) general techniques for program optimization, including the efficient representation of knowledge about programs and the use of measurements to guide analysis, and (2) particular code optimization strategies to enhance the use of EL1 as an implementation language.

The general optimization techniques we have studied are now being applied in two settings.

(a) An interactive program transformation facility called the Structured Programming Laboratory (SPL) is now being designed. The SPL will address the issues of optimization at high levels of language, precise control of tools and visibility of their results, and the use of manifest, measured, and user provided knowledge about programs to assist their optimization.

(b) An optimization phase is being added to ECL´s interpreter-compatible compiler. Although designed to operate under strict resource constraints, and required to be consistent with the full power of ECL´s data structure and control definition facilities, the optimizer we have developed incorporates some interesting techniques for detecting and removing the sorts of redundancy that arise most frequently.

To experiment with the use of EL1 as an implementation language, a compiler for a systems programming dialect (SPECL) is being constructed. The SPECL project complements work on the SPL, since it is concerned primarily with optimizations at the machine level: reordering computations to minimize temporary storage, register allocation and assignment, and the like.

Section II of this report gives an overview of how optimization at the source level will be performed using the SPL. Sections III and IV discuss aspects of the representation of facts about programs and the use of measurements to aid code improvement. Section V describes the new optimization phase of the compatible compiler, and Section VI discusses the SPECL compiler.

II. Optimization in the Structured Programming Laboratory

There seems to be a "critical mass" phenomenon in program optimization: the practicality of a single tool is greatly enhanced by the existence of others. Such classical loop optimizations as strength reduction and test replacement, for example, may give rise to "dead" variable assignments that must be removed. Open-substitution of procedures, i.e. the replacement of procedure calls by equivalent expressions derived from the procedure bodies, leads to opportunities for redundancy elimination. In short, the whole often seems more effective than the sum of the separate parts. Our studies of program specialization and measurement-assisted compilation have repeatedly indicated that to forge new tools one must rely on an arsenal of existing techniques.

Thus, a principal aim of our recent work has been the design of the Structured Programming Laboratory, a system based on ECL that will both incorporate the best of our existing program development and manipulation tools and serve as the setting for future work on program improvement.

To its users, the SPL will act like a program editor with special facilities to help manage the ebb and flow of program development from abstract specification to concrete realizaton. As successive refinements take shape, they will be recorded in a development history. The history will exhibit the global structure of developing systems and display the logical connection among decisions made during design. The explicit retention of this structure not only aids understanding and maintenance of the program, but can be used to support optimization during refinement as well.

Suppose a program to manipulate sets makes use of a special syntactic structure to iterate over the elements of a set. An example might be

```
    FOR\EACH E IN SUCCESSORS(H) DO
      F(E) ->
        S <- S UNION {E}
```

This iterator might be implemented by a rewriting rule or "syntax macro" such as

```
FOR\EACH E IN  $S  DO  $B  -->
     BEGIN
       DECL TEMP:SET BYVAL  $S;
       REPEAT
         EMPTY(TEMP) => EXIT;
         DECL E:ELEMENT LIKE NEXT(TEMP);
         $(B)
       END;
     END;
```

In a conventional system each call of this macro would be expanded, and the set primitives EMPTY and NEXT perhaps further expanded before optimization could begin. In the SPL, it will be possible first to study the improvement of the macro body in isolation. For instance, storage for the element E might better be allocated once outside the loop:

```
          DECL E:ELEMENT;
          REPEAT
             EMPTY(TEMP) => EXIT;
             E <- NEXT(TEMP);
             $(B);
          END
```

Nor is it necessary that every call be expanded in-line. The optimizer might generate a subroutine from the macro body, taking a SET $S and a parameterless function $B as arguments. Or it might mix the open and closed implementations depending on estimates of the relative importance of the points of call.

Although quite simple, this example shows the importance of not throwing away useful structural information only to try to recover it later through extensive global analysis. Recognizing and merging instances of this little set mapping routine, using an expand-then-optimize strategy, might become an expensive task given sizable bindings for $B.

We further feel that the approach of involving optimization at high language levels can lead to a kind of bootstrapping effect. Where extended operators (such as FOR\EACH) are frequently used in programs written in high level terms (like the SET domain), it will be profitable to pre-analyze their definitions, singly or in typical combinations, to identify _a priori_ the special conditions under which simplifications can be made. In the FOR\EACH case, the loop body can be reordered if the argument SET $S is not initially EMPTY. If $S is completely known at the point of call, it may not be necessary to generate a loop at all. In short, it may be possible to extend the SPL´s general optimization algorithms to special domains, and to minimize the cost of translating programs over those domains.

III. Knowledge About Programs - The Context Graph

Useful manifest information is sometimes overlooked by conventional optimizers because its discovery requires deductive power beyond that of the analyzer or because maintaining sufficient facts about the dynamic execution states of a program appears too expensive.

Consider the sequence

```
1 ≤ I AND I ≤ LENGTH(A) -> [) ... A[I] ... (]
```

Few compilers would take cognizance of the predicates involving I to eliminate range checking when the selection A[I] is compiled. Yet well-written, defensive programs often include just such checks on data, making the further checking required by language semantics redundant. There is a host of similar situations in which optimizers miss important semantic features because of the difficulty or expense of bookkeeping.

The heart of the SPL will be a Symbolic Evaluator. Part of its job will be to manage program knowledge to make simplifications possible and efficient. While the design of the Evaluator is not yet fixed, we will describe a method we have developed by which context-dependent facts can be maintained with little or no storage duplication and only modest access overhead.

Facts such as those about the value of I in the example above will be stored in a data base, accessible by hashing the names of items they describe. Each fact will be tagged with a context descriptor. We use the term context to denote a program region in which control is strictly sequential. A context graph is a labeled, acyclic, directed graph whose nodes represent the contexts (flow blocks) of the program. The context labels are integers assigned in evaluation order: if node Ni can precede node Nj during some computation (ignoring loop traversal), then LABEL(Ni) < LABEL(Nj). The arcs in a context graph are of two kinds: possible predecessor (PP) arcs and essential predecessor (EP) arcs. Each node N except the unique entry node has one EP arc pointing to its immediate flow dominator, the nearest context through which control must always pass to reach N. Each node has a set of PP arcs pointing to the contexts that may immediately precede it in some computation.

To show how the context discrimination works, we will give an algorithm for deriving the set of possible values for some variable V at a particular program point C, given the set of value-facts about V for the whole program.

Let G be the context graph of the program with E its unique entry node. Define a marking of a context graph as an ordered subset of the nodes, in decreasing order of label. The set of contexts in which program variable V acquires new values will be given by a marking M of graph G.

The problem is to derive a submarking S from M that satisfies two conditions with respect to the particular context of interest C:

(1) each member of S can be reached via PP arcs from C without passing members of M, and

(2) every PP path from C to E (the unique entry node) contains at least one member of S.

The members of S correspond to bindings of V that can still be in effect when control reaches C during program execution.

A simple scheme for computing S can be based on the connectivity matrix P defined by

P[I, J] = the number of PP paths in G from the node labeled I to that labeled J.

P can be computed (once, in advance) using Warshall´s algorithm with arithmetic sum and product operations. Or it can easily be developed "on the fly" as context labels are assigned during initial analysis.

To see how S can be obtained efficiently using P, imagine an intermediate stage of the computation at which a partial marking S has been produced. That is, members of S (L in number) have been selected from M in decreasing label order, with those not satisfying condition (1) above rejected. The process is to continue until condition (2) is also satisfied. Suppose also that associated with the Kth member of set S is an integer, NUMBER\CLEAR[K], which gives the number of PP paths from C to the entry E that pass through S[K], but not through any S[J], for $1 \leq J < K$.

The next member of S is found by examining successive candidates T from M until one is found having at least one clear PP path (i.e., unmarked by members of M) from C, or until M is exhausted. Note that any marked path from C to T must contain at least one S node, since any marker from M that could occlude T from C will already have been considered. So it suffices to count paths to T that are clear of S nodes.

Let NUMBER\TO\T be P[LABEL(C), LABEL(T)], the total number of paths from C to T. Since the labels along every such path must decrease, the number of clear paths is

```
FOR K TO L REPEAT
 NUMBER\TO\T <+
  - P[LABEL(S[K]), LABEL(T)] * NUMBER\CLEAR[K]
END
```

where <+ is an operator that increments its left operand by its right, and the product computed on the Kth iteration represents the number of paths from C to T that begin with clear paths from C to S[K].

If there are clear paths to T, then T is inserted at the shallow end of S and NUMBER\CLEAR[L+1] is installed. Since T occludes some new paths from C to E, the entry of G, a counter is decremented to determine whether every path to

E has been blocked. If so, the program halts with S as its result.

The following ECL procedure computes the submarking of M that occludes node C, if one exists. It is assumed that each member of M has a lower label than LABEL(C) (i.e., potential values for V lying past C will be discarded outright), and that the connectivity matrix P has already been computed.

```
EXPR(M:MARKING, C:NODE; MARKING)
 < MARKING, "C BLOCKED" > <<
  BEGIN
   DECL NUMBER\CLEAR:SEQ(INT) SIZE CARDINALITY(M);
   DECL NUMBER\TO\E:INT BYVAL P[LABEL(C), LABEL(E)];
   DECL NUMBER\TO\T:INT;
   DECL S:MARKING;
   DECL L:INT                     /* ´CARDINALITY OF S´;
   FOR\EACH T IN M
    DO BEGIN
     NUMBER\TO\T <- P[LABEL(C), LABEL(T)];
     NUMBER\TO\T = 0 => NOTHING   /* ´NO PATHS FROM C TO T´;
     FOR K TO L
      REPEAT
       NUMBER\TO\T <+
        - (NUMBER\CLEAR[K] * P[LABEL(S[K]), LABEL(T)]);
      END;
     NUMBER\TO\T = 0 => NOTHING   /* ´T MARKS NO NEW PATHS FROM C´;
     APPEND(S, T)                 /* ´INSERT T AT THE END OF S´;
     NUMBER\TO\E <+
      - (NUMBER\TO\T * P[LABEL(T), LABEL(E)]);
     NUMBER\TO\E = 0 -> RETURN(S, "C BLOCKED");
     NUMBER\CLEAR[L <+ 1] <- NUMBER\TO\T;
    END;
   S;
  END;
```

The algorithm can be refined by making use of the essential predecessor information in the context graph.

Obviously, C will be blocked as soon as one of its flow dominator nodes is blocked. So for each trial node T we need only consider paths to T from the earliest essential predecessor of C whose label exceeds LABEL(T).

The changes to the procedure are straightforward. C is bound BYVAL so that it can be rebound to the least back-dominator of the input C as each candidate T is chosen. (We assume that EP(E) points to a special node whose LABEL is less than any other.) A special check is made for the case in which T is one of C´s essential predecessors. TAIL, a new local INT, will index the first element of S installed since C´s last adjustment. Previously discovered S nodes cannot possibly block the new C; hence, they are disregarded by the summation over blocked paths from C. That loop becomes FOR K FROM TAIL TO L REPEAT ... . The full algorithm is

```
EXPR(M:MARKING, C:NODE BYVAL; MARKING)
 < MARKING, "C BLOCKED" > <<
  BEGIN
   DECL NUMBER\CLEAR:SEQ(INT) SIZE CARDINALITY(M);
   DECL NUMBER\TO\E:INT BYVAL P[LABEL(C), LABEL(E)];
   DECL NUMBER\TO\T:INT;
   DECL S:MARKING;
   DECL L:INT                        /* ´CARDINALITY OF S´;
   DECL TAIL:INT BYVAL 1             /* ´TAIL OF S FOR CURRENT C´;
   FOR\EACH T IN M
    DO BEGIN
     LABEL(EP(T)) GT LABEL(T) ->
      BEGIN
       REPEAT
        C <- EP(C);
        LABEL(EP(C)) LT LABEL(T) => NOTHING;
       END;
       C = T =>
        [) APPEND(S, T); RETURN(S, "C BLOCKED") (];
       TAIL <- L + 1                 /* ´EARLIER S MEMBERS CANT HIDE C´;
       NUMBER\TO\E <- P[LABEL(C), LABEL(E)];
      END;
     NUMBER\TO\T <- P[LABEL(C), LABEL(T)];
     NUMBER\TO\T = 0 => NOTHING      /* ´NO PATHS FROM C TO T´;
     FOR K FROM TAIL TO L
      REPEAT
       NUMBER\TO\T <+
        - (NUMBER\CLEAR[K] * P[LABEL(S[K]), LABEL(T)]);
      END;
     NUMBER\TO\T = 0 => NOTHING      /* ´T MARKS NO NEW PATHS FROM C´;
     APPEND(S, T)                    /* ´INSERT T AT THE END OF S´;
     NUMBER\TO\E <+
      - (NUMBER\TO\T * P[LABEL(T), LABEL(E)]);
     NUMBER\TO\E = 0 -> RETURN(S, "C BLOCKED");
     NUMBER\CLEAR[L <+ 1] <- NUMBER\TO\T;
    END;
   S;
  END;
```

The computed submarking S gives the set of possible values for V at C as accurately as can be determined from static analysis. Note that the speed of the algorithm depends not on the size of the graph G, but only on the size of the initial set of possibilities. For a typical variable, this set is likely to be small.

IV. Uses of Measurements in Optimization

Execution profiles have been recognized for some years as helpful aids to the programmer. They guide his attention to the small but critical inner loops where programs typically spend most of their time. And profiles highlight sections of the program not being executed either because of bugs or because of inadequate test data.

In the SPL, timing measurements and frequency counts taken from representative sample data will aid not only human users but will help drive refinement and optimization tools as well. Because metering probes are simply ECL data items, they can be included along with other program knowledge in the SPL data base. Because ECL permits them to be placed and activated with great selectivity, they can give a clearer picture of data attributes and program behavior than conventional profiles.

Measurements act both as a guide to important program regions and as a rein on the analyzer. It can be permitted to expend most effort where the potential payoff is greatest. Comparison of frequency counts may lead to reordering of branches, when that can be done safely, and frequencies enable the decision whether to expand a macro or call a subroutine to be made selectively.

The fact that a region is not executed at all under certain conditions may suggest simplifications to be verified either by the Symbolic Evaluator or by asking the user. This may be especially fruitful when a very general procedure is being applied to a special case task. Consider, for example, this doubly recursive list structure copy routine:

```
COPY =
   EXPR(A:LIST; LIST)
     BEGIN
        ATOMIC(A) => A;
        ALLOC(LISTCELL OF
              COPY(HEAD(A))     /* 1,
              COPY(TAIL(A))     /* 2);
     END
```

Suppose that COPY is called in an iteration whose skeleton is

```
   REPEAT
       ...
       L <- READLIST();
       ...
       COPY(L)          /* 3;
       ...
   END;
```

In the SPL it will be possible to plant metering probes that discriminate among several points of call of a function. After being metered and invoked by its calling program on 100 sample input lists, the COPY procedure, printed with its probes, might look like:

```
   EXPR(A:LIST; LIST)
     BEGIN
       <300, 300, 100> ATOMIC(A) => <300, 100, 0> A;
       <0, 200, 100> ALLOC(LISTCELL OF
                           COPY(HEAD(A))         /* 1,
                           COPY(TAIL(A))         /* 2);
     END
```

The elements of the frequency tuples <c1, c2, c3> correspond respectively to the labeled points at which COPY is called. The zero counts are interesting and can be used as cues by the Symbolic Evaluator. The fact that ATOMIC(A) is never FALSE when COPY is called from point 1 suggests that HEAD(A) is never a list; that is, that input lists will always be linear. If the user confirms this guess, then COPY can be significantly simplified: its first recursive call can be replaced by HEAD(A). The fact that ATOMIC(A) is never TRUE when the point of call is 3 suggests that input lists L are never empty. If this were also confirmed, it might be advantageous to avoid one test per call by replacing COPY(L) with ALLOC(LISTCELL OF HEAD(L), COPY(TAIL(L))).

An important feature of the SPL will be an implementation library, a collection of packages oriented towards specific problem domains, each containing syntactic extensions, modes and mode gerators, debugging aids, and sets of alternative data representations and access algorithms. Use of preplanted metering probes by these packages should be a most effective way of providing machine assistance during implementation, since a package can be built to interpret its own probes intelligently. A sorting package, for example, can collect statistics during development and testing of a user´s program and then give him quantitative advice on which strategy to select for the production version. A sparse matrix package can gather measurements not only on the patterns of sparseness of matrix elements but on the accessing sequences most commonly used. It can then offer the user an assessment of the space/time trade-offs involved in his particular application.

V. <u>Optimization Phase in the ECL Compatible Compiler</u>

While the SPL will help its users eliminate or avoid major redundancy at the source level, it should not be forced to carry the process too far. A little redundancy often makes a program easier to read. Straightforward optimizations not requiring the full machinery of the SPL should properly be included in the machine code generator.

We have therefore designed and are now coding an optimization pass for ECL's compatible compiler. Called Pass 1.5, it walks the program tree output by Pass 1 (the analysis phase) and produces a modified tree for input to Pass 2 (the code generation phase). To permit the compiler to remain usable within limited resources, we have chosen to concentrate on the problem of eliminating redundant subexpressions, and on the so-called "invalidation problem" for potentially available expressions.

The invalidation problem arises because sharing patterns among variables and data structures permit hidden side effects to destroy the availability of expressions not manifestly affected. For example, suppose X and Y are LIST variables, and consider the sequence

```
        ...  HEAD(TAIL(X)) ...
           Y <- NEWLIST;
        ...  HEAD(TAIL(X)) ...
```

Unless it can be shown that Y is shared neither with X nor with TAIL(X), the value of HEAD(TAIL(X)) must be recomputed; that is, it is invalidated by the possible side effect. In a language with the rich potential for sharing that ECL has, the relationship between X and Y may be quite obscure, particularly if they are formal parameters or free variables of the procedure being compiled. This is sometimes called the "aliasing problem."

Our approach in Pass 1.5 is to cope as effectively as we can with the invalidation problem without engaging in intense analysis. Fortunately, a significant number of potentially damaging side effects can be ruled out on very simple grounds of scope, storage status (heap versus stack) or data type.

Two types of common subexpressions are distinguished: pure values and proper objects. Pure values are expressions like X + Y and LENGTH(A) whose values are transient; that is, they occupy no identifiable place in storage. The availability of a pure value for reuse ends if any of is subexpressions is redefined. When this type of redundant expression is removed, its replacement acts like a "runtime constant": storage for the value is allocated at procedure entry and is initialized when the "parent" or defining occurrence is evaluated. Thereafter it may be referenced but not changed.

Proper object expressions are those with identifiable, reusable storage locations, such as A.FIELD[I] and VAL(P). When found redundant, they too are replaced by references to temporaries. But proper object temporaries behave like shared variables, rather than constants. For example, the sequence

```
        DECL A:RECORD;
             ...
        PRINT(A.FIELD[I]);
             ...
        A.FIELD <- NEWFIELD;
        PRINT(A.FIELD[I]);
```

would be replaced, in effect, by

```
        DECL A:RECORD;
             ...
        DECL T1:ANY SHARED A.FIELD;
        DECL T2:ANY SHARED T1[I];
        PRINT(T2);
             ...
        T1 <- NEWFIELD;
        PRINT(T2);
```

Care must be taken to see that stack space for proper object temporaries is allocated at the same block level as the parent object (RECORD A in the example above). If it were allocated at the procedure level, for example, dangling references could outlive an object allocated on the stack, leading to confusion for the storage manager.

As Pass 1.5 walks the program tree, it maintains an Available Expression List (AEL). Only simple expressions using built-in operators are included; control expressions are not. Each expression listed in the AEL is classified by mode, by context of creation, by expression type (pure or proper), and if proper, by scope and locale (local versus global, stack, heap or unknown). As a new simple expression is walked, it is tested for redundancy with existing available expressions. Two expressions match when

(1) they have the same operator, and
(2) they have the same number of operands, and
(3) each pair of corresponding operands match recursively.

If the new expression matches none of the AEL members it is made available by adding it to the list. Otherwise it is linked to the existing expression subtree.

Various transitions and events in Pass 1.5´s walk of the tree cause it to scan the AEL and prune invalid elements. A change of context, for example, may cause some expressions to become "unavailable" while the validity of others is re-established. A control excursion, such as a call to a procedure whose effects are unknown, forces all available expressions to be dropped except those depending strictly on "hidden" local names. Pruning expressions at an assignment is essentially a pattern matching process, using the object descriptions stored with AEL members. When the object being assigned is well defined, e.g. an unshared local variable of known mode, affected expressions can be pruned quite precisely. In less well resolved cases of course, more AEL elements may have to be removed to guarantee correctness. For example, VAL(INTPTR) <- I + 1 invalidates not only the expression being redefined but also all INT expressions not known to reside on the stack.

Even so, it appears that the inexpensive classification scheme used in Pass 1.5 will provide a most satisfactory and efficient adjunct to the more powerful tools of the SPL.

VI. The SPECL Compiler

The SPECL (Systems Programming in ECL) project is intended to extend the use of ECL into areas normally reserved for so-called "implementation languages." It is a dialect of ECL that can be compiled to stand-alone machine code that runs without the support of ECL´s runtime facilities. It offers the opportunity to "contract" ECL for special applications, since SPECL-produced code can be augmented by just the runtime support (such as storage management or I/O) that´s needed.

SPECL also offers access to the implementation of operators and the choice of underlying data representations at the hardware level. For example, suppose a programmer wants to implement doubly linked lists using a minimum of storage for the links. One trick is to give each element a single link field containing the bitwise exclusive-or of the address of its predecessor with that of its successor. Given pointers to any two successive elements it is then a simple matter to move forward or backward along the list.

SPECL is ideal for such an application since it permits the user to manage storage as he chooses and allows him to give machine code definitions for the necessary pointer operations. Access to the hardware level is isolated in code generation templates called Compiler Control Expressions (CCE). CCE´s tell the compiler how to implement a given operator, depending on the states of its operands. The descriptions of operand states serve as goals for the code generator as well as enabling code selection. The CCE´s contain enough information about register use and side effects that optimization of expressions containing user-specified operators need not be interrupted.

The compiler consists of three phases. The first labels the program tree with temporary storage requirements (for subtrees free of common subexpressions (CSE)) in a manner similar to the Sethi-Ullman method. Other properties that can be deduced at this time are brought out and attached to the tree. An initial pruning of the possible code generation templates takes place as well.

The second phase operates on a partially flattened version of the tree. It reorders computations to try to minimize the use of temporary storage. This step is necessary because the Sethi-Ullman algorithm does not allow for CSE´s. The method is heuristic and incomplete because the minimization problem is polynomial complete.

Register allocation and assignment are also performed during the second phase. Like minimization, the assignment problem is polynomial complete, and it will be handled heuristically.

The third phase is straightforward code generation.

Starting from a trial ordering of the expressions in each context (straight-line program section), the temporary-minimization procedure examines the variation in storage requirements over each context. The vicinities of peaks in the requirements are scanned for target positions that meet a simple numeric criterion based on the temporary usage and result size of the neighboring computations. Expressions are moved to these favorable positions in order to reduce overall storage use. CSE-free subtrees move as units, subject to safety contraints. Most of the time used by this process is actually spent in recalculating the temporary requirements after a move.

The register assignment algorithm first makes a live-dead analysis of variables and common sub-expresseions and determines the minimum distance to next use for these items. This gives the information needed to perform register allocation optimally in straight-line code. Assignment is straightforward in such branch-free regions as well. With control structure, however, the problem is to match assignments at branches and join points. Heuristics are needed to reduce the computation from a "try all assignments" approach. At present, we have only a simple algorithm for this problem. As usual, the issue is the trade-off between the cost of the algorithm and the improvements in the code it makes.

In summary, then, SPECL will extend to the hardware level the methodology that will characterize use of the SPL. Users will be permitted to become involved in the optimization of their programs, and they will not need to forsake good structure to achieve highly efficient performance.